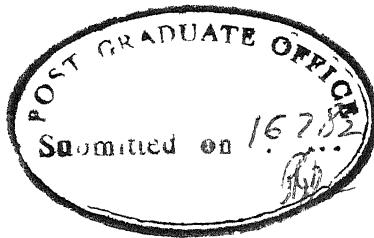


ON SEMANTICS OF FUNCTIONAL PROGRAMMING

A Thesis Submitted
In Partial Fulfilment of the Requirements
for the Degree of
MASTER OF TECHNOLOGY

by
G. S. KUMAR

to the
COMPUTER SCIENCE PROGRAMME
INDIAN INSTITUTE OF TECHNOLOGY, KANPUR
JULY, 1982



CERTIFICATE

This is to certify that the thesis entitled, 'On Semantics of Functional Programming' has been carried out by G.S. Kumar under my supervision and has not been submitted elsewhere for the award of a degree.

S Biswas

Dr. S. Biswas
Lecturer

Computer Science Dept.
Indian Institute of Technology
Kanpur

Kanpur:

July 1982:

J 1984

ITRI LIBRARY
I T R I - A P U R
A.....82661

CSP-1982-M-KUM-ON

ACKNOWLEDGEMENT

I express my deep sense of gratitude to Dr. S. Biswas for his inspiring guidance, patient understanding and constant encouragement.

I also wish to thank all my friends who made the stay at I.I.T. Kanpur a pleasant experience.

Lastly, I would like to thank Mr. M.C. Gupta for his typing of the thesis.

(G.S. KUMAR)

Kanpur:

July 1982:

ABSTRACT

Functional Programming systems introduced by Backus have, as their chief advantage, attractive algebraic properties. This thesis proposes to make use of these properties for the purpose of specifying the semantics of FP systems. The central idea of the thesis is that a set of equations to be satisfied by the operators (combining forms) of FP system provides the transformational semantics. Also it is proposed that this set of equations can be thought of as formal system for deducing program equivalence. One such set of equations is proposed and studied. The program equivalence proofs derived in this system are presented here and these show that the system is easy to use and yet powerful.

CONTENTS

<u>Chapter</u>	<u>Title</u>	<u>Page</u>
I	Introduction	1
II	FP System	4
III	Semantics of FP	11
IV	Examples	25
V	Conclusions and Suggestions for Further Work	42
	References	

CHAPTER 1

INTRODUCTION

The Functional Style of programming [Ba 78] has been introduced by Backus as an alternative to the conventional style of programming. Functional Programming, hereafter referred simply as FP, systems essentially consist of a set primitive functions and combining forms. Combining forms are used to build new programs from existing programs. It is these combining forms which are mainly responsible for the ease of programming in an FP system. The main points of differences between conventional and Functional style are:

i) Use of variables:

Conventional programming languages have variables and the notion of assigning values to variables. This necessitates usually complex state transition functions to provide the semantics of such languages. FP, on the other hand, is variable free, and therefore its semantics can be given in a simpler way.

ii) Use of combining forms:

The rich variety of combining forms that can be used to construct new functions from old ones in FP is almost totally absent in conventional programming. This precludes any possibility of developing an algebra for conventional programming languages.

For a more detailed and lucid development of FP we refer to [Ba 78].

The combining forms of an FP system are so chosen that they enjoy useful relationships among themselves and these can be expressed as algebraic identities or equations. Thus in FP it is possible to develop an algebra for reasoning about programs by manipulating or transforming programs themselves with the help of these equations.

We propose to utilise these algebraic properties for specifying the semantics of FP. Our notion of semantics is that the class of equivalent programs (programs computing the same function) define the function itself. In this view, providing semantics reduces to finding the equivalence class of a program. As we have mentioned above, FP programs can be transformed into one another through the use of the algebraic identities satisfied by the operators. Thus if we can find a set of equations which is strong enough to specify all possible transformations among programs, then the equivalence relation generated by this set of equations will constitute a proper semantics. Moreover, the same set of equations can serve as a formal system for program equivalence.

We have attempted to provide in this thesis such a set of equations. We have not been able to characterize the power of this set of equations, that is we do not know whether from this set of equations we can obtain an equivalence proof of any two programs that will be regarded intuitively equivalent. However we have been able to derive a number of

equivalence proofs within this system and this fact may be reckoned towards the practical usefulness of the system. We have not formally proved the soundness of the system (that is a demonstration that no two non-equivalent programs will have a proof of equivalence in the system) as the intuitively self-evident nature of the equations render soundness obvious.

In the next chapter we provide informal introduction to an FP system to be used in this thesis. Chapter 3 contains the development and discussion of our equational system. Chapter 4 presents some of the interesting proofs obtained using this system. Chapter 5 presents conclusions of the present work and suggestions for future work.

CHAPTER 2FP SYSTEMS

In this chapter we provide an informal view of an FP system which will be used in this thesis. Primitive functions of the system are familiar ones, whereas the combining forms are simple enough to be understood intuitively from their informal descriptions, which we provide here. In the next chapter we shall see how we propose to formalize this intuitive understanding.

1. FP System:

An FP system consists of:-

- a) Set A of atoms , including \emptyset , the null sequence
- b) Set O of objects defined recursively by
 - i) \perp is in O (the undefined object)
 - ii) A belongs to O (all atoms are objects)
 - iii) If x_1, \dots, x_n are in O then $\langle x_1, \dots, x_n \rangle$ is in O
(all sequences are objects)
 - iv) $\langle x_1, \dots, x_n \rangle = \perp$ if for some i , $x_i = \perp$
- c) Set P of primitive functions and set T of combining forms and a function $\text{Arity}: T \rightarrow N$ where N is set of natural numbers. Arity gives the number of arguments a particular combining form takes.
- d) Set F of function defined recursively by
 - i) P is a subset of F (all primitive functions are functions)

- ii) If f_1, \dots, f_n belong to F and C belongs to T and $\text{Arity}(C)=n$ then $C(f_1, \dots, f_n)$ is in F .
- iii) The function defined by $\text{Def } g = E$ is in F if E is a combining form constructed by applying combining forms chosen from T to functions chosen from F together with the function symbol g .
- e) The concept of application of function f in F to an object x in O denoted by $f:x$

The sets P and T will be so chosen that all f in F are strict, i.e., ' $f: \perp = \perp$ '.

A program in FP is an expression representing a function. All functions are of one type only, namely $[0 \rightarrow 0]$. All functions take single arguments only. There are no variables in FP and combining forms are the means of building new functions from existing ones.

2. Primitive functions and combining forms:

We now describe a particular FP system which will be used in the sequel.

We assume that set of atoms A contains integers, booleans and strings over some alphabet.

Below we give a list of primitive functions and their informal meanings. Some of these functions can be expressed in terms of other functions but in this chapter

we treat them all as primitive. In the next chapter we use only the essential primitives as primitive functions.

<u>Name</u>	<u>Description</u>
Hd, Tl	Head and Tail functions (same as CAR and CDR of LISP)
id	identity function , $id: x=x$ for all x in 0 and $x \neq \perp$
Appendl, Appendr	Append function of LISP where only left or right list respectively, is expanded. Appendl: $\langle y, \langle z_1, \dots, z_n \rangle \rangle = \langle y, z_1, \dots, z_n \rangle$ Appendr: $\langle \langle z_1, \dots, z_n \rangle, y \rangle = \langle z_1, \dots, z_n, y \rangle$
Distl Distr	Distl: $\langle y, \langle z_1, \dots, z_n \rangle \rangle = \langle \langle y, z_1 \rangle, \dots, \langle y, z_n \rangle \rangle$ Distr: $\langle \langle z_1, \dots, z_n \rangle, y \rangle = \langle \langle z_1, y \rangle, \dots, \langle z_n, y \rangle \rangle$
Trans	Trans: $\langle \emptyset, \dots, \emptyset \rangle = \emptyset$ Trans: $\langle x_1, \dots, x_n \rangle = \langle y_1, \dots, y_n \rangle$ where $x_i = \langle x_{i1}, \dots, x_{im} \rangle \quad 1 \leq i \leq n$ and $y_j = \langle x_{1j}, \dots, x_{nj} \rangle \quad 1 \leq j \leq m$
Length	Same as LISP Length function
$+, -, *, \div$ And, or, not	Usual operators of Arithmetic and Logic
Atom, Eq, Null, Le, ..	Usual predicates

This completes the list of primitive functions we will be using in this thesis.

The following set of combining forms are used.

1. Composition: This is the usual composition of functions and is denoted by \cdot as infix operator.

$$(f \cdot g): x = f:(g:x)$$

2. Construction: denoted by $[...]$

$$[f_1, \dots, f_n]: x = \langle f_1:x, \dots, f_n:x \rangle$$

3. Conditional: Similar to COND of LISP and is denoted by \rightarrow .

$$(p \rightarrow f; g): x = \begin{aligned} &\text{if } (p:x) = \text{true} \text{ then } f:x \\ &\text{else} \\ &\quad \text{if } (p:x) = \text{false} \text{ then } g:x \\ &\text{else } \perp \end{aligned}$$

4. Constant: For every atom x in Λ there is a constant function denoted by \bar{x} .

$$\bar{x}:y = \text{if } (y = \perp) \text{ then } \perp \text{ else } x.$$

5. Insert: This is similar to REDUCE operator of APL and is denoted by $/$.

$$(\text{/ } f): \langle x_1, \dots, x_n \rangle = f: \langle x_1, (\text{/ } f): \langle x_2, \dots, x_n \rangle \rangle \quad n \geq 2$$

$$(\text{/ } f): \langle x_1 \rangle = x_1$$

6. Apply to All: denoted by α .

$$(\alpha f): \langle x_1, \dots, x_n \rangle = \langle f: x_1, \dots, f: x_n \rangle \quad n \geq 1$$

$$(\alpha f): \emptyset = \emptyset$$

This completes our description of the FP system we will be using. In the next section a few examples to illustrate how to write or build programs in this FP systems are provided. For more example we refer to [wi 81].

3. Sample Programs:

As our first example we provide the matrix multiplication program. We define IP the inner product of two vectors.

Def IP = $(/+) \cdot (\alpha *) \cdot \text{Trans}$

The program captures the essential idea that corresponding coordinates of the vector have to be multiplied and summed up and no more than this.

Next let us choose to represent a matrix by a sequence of rows each row being a vector. Thus matrix M would look like

$$M = \langle m_1, \dots, m_r \rangle \text{ where } m_i = \langle m_{i1}, \dots, m_{ic} \rangle$$

where r is the number of rows and c the number of columns. Let M and N be two compatible matrices for multiplication.

We define the matrix multiplication program MM as follows:

Def $MM = (\alpha \alpha IP) \cdot (\alpha DISTL) \cdot DISTR \cdot [Hd, TRANS \cdot 2]$

where 2 is a selector function to pick the second element of a sequence. This program again brings out the strong point of FP, that of stating what is essential and nothing more. What this definition states is that with each row of Matrix M all the columns are associated and their inner product is taken. This program also brings out the power of combining forms to build programs.

As our second example we take slightly more complicated problem. We want to find the connected components of a graph. Let the graph be provided as the incidence matrix of 0's and 1's. This matrix is symmetric and reflexive. Treating this matrix as a relation, taking transitive closure will give us the required components as equivalence classes.

Thus we define the function TC for Transitive closure as follows

Def $TC = (/MM) \cdot COPY \cdot [Length, id]$

What is being done make n (number of vertices in the graph) copies, using the function COPY (which can be defined in a straight forward manner), of the incidence matrix and multiply them together. We can define CC the connected component program as follows

Def $CC = Compact \cdot \alpha Rowclass \cdot DISTR \cdot [id, id] \cdot TC$

Take the transitive closure of the given graph. In this equivalence relation find the equivalence class for each vertex. Then pick those equivalence classes which are distinct. Function Rowclass finds the equivalence class of a particular vertex. Rowclass may be defined as:

Def Rowclass = Makeclass • [α Eqvector.DISTR, I]

Eq vector is just a function to test equality of vectors and is straightforward. α Eqvector.DISTR is boolean vector with T for row belonging to the same class. Makeclass function just picks up these row elements. The rest of the functions are easy to define.

These two example should suffice to convince that an FP system is powerful and also programs can be built hierarchically. A few more examples can be found chapter 4.

CHAPTER 3

SEMANTICS OF FP

In this chapter we present our approach to the Semantics of FP. As a by-product we also get a proof system. We start by explaining our view of semantics and explain how a system of Equations can be regarded as providing 'Semantics'. Such a system of equations can also be thought of as a formal system to carry out proofs to show that two programs are equivalent. We shall present here such a system of equations in an attempt to completely specify the FP system semantics. We shall then discuss the merits of and the problems associated with our choice. We end the chapter by mentioning general properties of such systems of equations and by giving references to related works.

1. Semantics:

Semantics of a programming language should enable one to understand, state and prove properties of programs written in that language. One approach to semantics deals with specifying the input-output behaviour of a program, i.e., the function (or relation if the language has non-determinism) computed by the program. Denotational semantics assigns one function to each construct of the language and these are defined through a mutual set of recursive definitions. The solution to this set is guaranteed by Scott's Theory. The central idea is to associate a function with a program which transforms the initial state to a final state.

We have outlined the above approach to contrast with the approach we take. Since programs compute functions, it is the set of computable functions (or a subset of it) which we want to specify. It is our thesis that the class of all programs which compute the same function effectively defines this function. This idea has been used in the algebraic specification of Abstract Data Types (ADT). To explain what we just said we elaborate on the ADT specification in more detail.

ADT is not just a collection or set of values but with a set of operators defined on these values. Hence the structure of ADT corresponds closely to algebra. The main problem is to provide meanings of these operators without any concrete representation. Consider any well-formed expression by these operators. These expressions evaluate to certain values of the ADT. Two different expressions may evaluate to the same value. Can we deduce this fact without actually evaluating them? The answer is yes. If we are given some identities that are satisfied by these operators then we may be able to transform one expression to the other. If the set of equations is sufficiently powerful, or 'Complete', then we should be able to derive that two expressions are equal without evaluating them.

Thus we can view that a set of equations specify the ADT in the following sense. The values of the ADT are the equivalence classes of expressions where two expressions are equivalent if they are equal by the set of equations.

Thus a semantics of the operators is provided by a set of equations which equate all the expressions representing the same data type value. This is our notion of semantics of programs as well. If we are able provide a system of equations by which we can deduce the equivalence of two programs then this set of equations constitute a semantics of the language. This may be properly called transformational semantics, but algebraic semantics is more widely used.

In the next section we show the similarity between ADT and FP programs and examine how a set of equations to be chosen.

2. FP as an ADT:

In the last chapter we explained how functional programs are built from primitive functions and combining forms. Thus functional programs are the well-formed expressions formed by these operators. The ADT is the set of computable functions with the structure given by these operators. Thus from the discussion of previous section it follows that if a set of equations is found we have a semantics of these operators. In the introduction we mentioned that these operators do satisfy certain identities. Actually they were chosen so that a rich set of identities exist. Thus we can certainly hope that we can provide a set of equations. Such a set is presented in another section. But before proceeding further we briefly point out certain salient points of this approach as applicable to FP.

A specific set of equations induce a particular equivalence relation, i.e., two programs are equivalent iff they can be proved equal using this set of equations. Different sets of equations define different equivalence relations and this corresponds to various types program equivalences. For example the empty set of equations define a trivial program equivalence, i.e. $P_1 = P_2$ iff they syntactically the same. The set of equations we have chosen is such that if two programs are equivalence by this set then they compute same output for all input values. Both programs will be undefined or they will be defined and equal for any input. Sometimes we may wish to have weaker equivalences. This would involve 'loosening' some of the axioms so that they can be used to deduce more equivalences. Thus one has to 'tune' a system to provide the type of equivalence one has in mind.

Another point to be mentioned is the form of equations. In actual practice each equation is not applicable for the whole range of functions. Thus we have to restrict the range of application of an equation. We have chosen to adopt the following. We prefix a predicate to an equation and the equation can be applied only when this predicate is true. Those equations not prefixed by any predicate are universally applicable. Also we have adopted a style for specifying these predicates which integrates with the FP style nicely. We explain this in the next section.

As in chapter 2 we assume that our FP system is strict, i.e. all functions evaluate to undefined if any one of their arguments is undefined. But this is not an essential feature of FP. Also in actual implementations using some

lazy evaluation schemes one is more interested in non-strictness for efficiency considerations. But strictness guarantees that any method of evaluation proceeds to obtain the least fixed point. Thus we have retained this condition though it does complicate some of the axioms by requiring them to be prefixed by predicates checking that all arguments are defined.

Another difficulty that has been faced is that FP functions are all of the same type, mapping objects to objects. But in practice any program written expects a particular type of input to be handled by the program. No elaborate checking to see that input has the required format and they are implicitly assumed to lead to undefined state. When performing program transformations one has to keep track of all these input requirements by the prefixed predicates. This makes the derivations cluttered with unnecessary details and the proofs look messy. The maintenance of the input types and their compatibility has to be done manually with adhoc arrangements. This is because there is no type specification mechanism in FP. To alleviate this problem to some extent we have introduced some type checking predicates like Issequence and Isnumber. In the absence any general mechanism for specifying types we have chosen only these two widely used types. But a more comprehensive type specification method would be very helpful.

In the next section we present a set of equations which will create an equivalence as defined above.

3. Equational Specification:

We name our system of equations as S. This set of equations has been chosen such that it is small and has no obvious redundancies. Also these axioms have certain intuitive significance as to what they define. This is explained in the next section.

We have grouped these equations as axioms and inference rules following the conventional usage of these terms. The axioms also have prefixed conditions but these are of trivial nature to keep track of some input conditions. But the conditions of inference rules generally use previously deduced results to derive new identity.

The notation we have adopted for specifying conditional equation need some explanation. The general syntax is:-

$p - \rightarrow g = d$

where p is a predicate, g and d are expressions. We have ambiguously used the truth values of FP (\bar{T} and \bar{F}) to denote Truth and False in our Meta-language as well. Also, we have used FP notation to describe the predicates wherever possible. As an example let us consider

Issequence.y \rightarrow Hd. Appendl. [x,y]=x. (A2)

Issequence.y is a predicate in FP. If the identity

$\bar{Issequence.y} = \bar{T}$ can be established then this Axiom can be used without prefixing this predicate to the equation being derived. Otherwise, we will prefix this

predicate to indicate the conditions underwhich the derived equation holds. One advantage is that the whole deduction can be carried out in FP language itself.

We are now ready to state the set of equations. All symbols are used as explained in chapter 2, where their meaning is intuitively explained. The set of Atoms is assumed to include integers and and Truth values.

Primitive functions:

- 1) Hd 2) Tl 3) Appendl 4) Length 6) Eq
- 7) Le 8) Atom 9) Null 10) Issequence 11) Number
- 12) Arithmetic operators 13) Logical operators

remark: The reason for not including other functions introduced in chapter 2 as primitive is that they can can be defined in terms of the above. Length is included in this list to facilitate the statement of some axioms, though it can also be defined from the above. Also the concept of length is fundamental.

In our set of equations we do not present axioms for arithmetic operators, logical operators as these are well known and we would like to use our familiarity with these operators more freely. The same for predicate LE as it depends on the axioms for arithmetic. The predicates Issequence and Isnumber also are not defined because their

intuitive meaning is clear and also they are used in defining equations and not in FP programs. But they can be given in a more elaborate systems.

Combining forms:

- 1) \cdot (Composition)
- 2) $[]$ (Const)
- 3) \rightarrow (Conditional)
- 4) $/$ (Insert)
- 5) α (Apply to all)
- 6) \perp (Constant)

Set of Equations (S):

AXIOM (A):

- A1. $\text{id} \cdot f = f ; f \cdot \text{id} = f$
- A2. $\text{Issequence} \cdot y \rightarrow \text{Hd} \cdot \text{Appendl} \cdot [x, y] = x$
- A3. $(\text{Issequence} \cdot y) \text{ and } (\text{T} \cdot x) \rightarrow \text{Tl} \cdot \text{Appendl} \cdot [x, y] = y$
- A4. $\text{Hd} \cdot [] = \perp ; \text{Tl} \cdot [] = \perp$
- A5. $\text{Appendl} \cdot [f_1, [f_2, \dots, f_n]] = [f_1, \dots, f_n] \quad n = 1, 2, \dots$
- A6. $f \cdot \perp = \perp ; [f_1, \dots, \perp, \dots, f_n] = \perp$
- A7. $\text{Length} \cdot [] = \overline{0} ;$
 $(\overline{1} \cdot x) \text{ and } \text{Issequence} \cdot y \rightarrow$
 $\text{Length} \cdot \text{Appendl} \cdot [x, y] = + \cdot [\overline{1}, \text{Length} \cdot y]$
- A8. $\overline{1} \cdot y \rightarrow \overline{x} \cdot y = \overline{x}$

intuitive meaning is clear and also they are used in defining equations and not in FP programs. But they can be given in a more elaborate systems.

Combining forms:

- 1) \cdot (Composition) 2) $[]$ (Const) 3) \rightarrow (Conditional)
- 4) $/$ (Insert) 5) α (Apply to all)
- 6) \perp (Constant)

Set of Equations (S):

AXIOM (A):

- A1. $\text{id} \cdot f = f ; f \cdot \text{id} = f$
- A2. Issequence $\cdot y \rightarrow \text{Hd} \cdot \text{Appendl} \cdot [x, y] = x$
- A3. (Issequence $\cdot y$) and ($\overline{T} \cdot x$) $\rightarrow \text{Tl} \cdot \text{Appendl} \cdot [x, y] = y$
- A4. $\text{Hd} \cdot [] = \perp ; \text{Tl} \cdot [] = \perp$
- A5. $\text{Appendl} \cdot [f_1, [f_2, \dots, f_n]] = [f_1, \dots, f_n] \quad n = 1, 2, \dots$
- A6. $f \cdot \perp = \perp ; [f_1, \dots, \perp, \dots, f_n] = \perp$
- A7. $\text{Length} \cdot [] = \overline{0} ;$
 $(\overline{T} \cdot x) \text{ and Issequence} \cdot y \rightarrow$
 $\text{Length} \cdot \text{Appendl} \cdot [x, y] = + \cdot [\overline{1}, \text{Length} \cdot y]$
- A8. $\overline{T} \cdot y \rightarrow \overline{x} \cdot y = \overline{x}$

A9. Atom . [] = \overline{T} ; Atom . $\overline{x} = \overline{T}$

$(\overline{T} . x)$ and (Issequence . y) Atom . Appendl . [x,y] = \overline{F}

A10. Eq . [id,id] = \overline{T} ; Eq . [x,y] = Eq . [y,x]

Eq . $[\overline{x}, \overline{y}] = \overline{F}$ for \overline{x} and \overline{y} distinct.

A11. Null . [] = \overline{T} ; Null . $\overline{x} = \overline{F}$ for \overline{x} different from [].
 $(\overline{T} . x)$ and (Issequence . y) \rightarrow Null . Appendl . [x,y] = \overline{F}

A12. Axioms of arithmetic and logical operators and axioms for predicates Le, Issequence, Isnumber.

A13. $f_1 . (f_2 . f_3) = (f_1 . f_2) . f_3$

A14. $[f_1, \dots, f_n] . h = [f_1 . h, \dots, f_n . h]$

A15. $[f_1, \dots, (p_i \rightarrow f_{i1}; f_{i2}), \dots, f_n] =$
 $(p_i \rightarrow [f_1, \dots, f_{i1}, \dots, f_n]; [f_1, \dots, f_{i2}, \dots, f_n])$

A16. $(p \rightarrow f_1; f_2) . g = p . g \rightarrow f_1 . g; f_2 . g$

A17. $g . (p \rightarrow f_1; f_2) = p \rightarrow g . f_1; g . f_2$

A18. $(\overline{T} \rightarrow f_1; f_2) = f_1$

$(\overline{F} \rightarrow f_1; f_2) = f_2$

A19. $\alpha f = \text{Null} \rightarrow []$; Appendl . [f . hd, (αf) . tl]

A20. $/f = \text{Eq} . [\text{Length}, \overline{l}] \rightarrow \text{Hd} ; f . [\text{Hd}, (/f) . tl]$

Inference Rules (I):

IR0. $(Hd \cdot f1 = Hd \cdot f2) \text{ and } (Tl \cdot f1 = Tl \cdot f2) \text{ and } (Issequence \cdot f1) \text{ and } (Issequence \cdot f2) \dashrightarrow f1 = f2$

IR1. $(p \dashrightarrow f1 = f2) \text{ and } (\text{not.}p \dashrightarrow f1 = f2) \dashrightarrow (\bar{T} \cdot p \dashrightarrow f1 = f2)$

IR2. $(p \dashrightarrow f = g1) \text{ and } (\text{not.}p \dashrightarrow f = g2) \dashrightarrow (\bar{T} \cdot p \dashrightarrow f = (p \rightarrow g1; g2))$

IR3. $(\text{Def } f1 = E1(f1)) \text{ and } (\text{Def } f2 = E2(f2)) \text{ and } (f1 = E2(f1)) \text{ and } (f2 = E1(f2)) \dashrightarrow f1 = f2$

IR4. $(\text{Isnumber} \cdot \bar{p}) \text{ and } (\text{Eq} \cdot [\text{size}, \bar{p}] \dashrightarrow f1 = f2) \text{ and } ((\text{Lt} \cdot [\text{size}, \bar{n}] \dashrightarrow f1 = f2) \dashrightarrow (\text{Eq} \cdot [\text{size}, \bar{n}] \dashrightarrow f1 = f2)) \dashrightarrow (\text{Ge} \cdot [\text{size}, \bar{p}] \dashrightarrow f1 = f2)$

This completes our list of equations. In the next two sections we briefly explain these equations and how to use them in practice.

4. Discussion of S:

Only a brief explanation of Axioms and Inference rules is attempted since most of them are self-evident. We note that some axioms are essentially schemas. For example

A10, in an actual system will be expanded into a series of axioms of the kind

Eq $[\bar{a}, \bar{b}] = \bar{F}$, where a and b are two distinct atoms of the system.

A2 - A5 define the data type of sequences. A6 is the strictness axiom. A8 defines the constant function with parameters from the set of atoms. A9 - A11 define the predicates. A14 - A17 show the interaction among the operators construction, composition and conditional. A18 gives the meaning of conditional. A19 - A20 define the combining forms α and $/$ respectively. Interaction of these operators with others is not given as axioms as they can be deduced from the defining equations.

Inference rule IR0 is a basic property of sequence operators Hd and Tl. IR1 and IR2 are two different versions of the well known case analysis. IR3 is the basic rule for dealing with recursive definitions and simply says that f_1 and f_2 defined by recursively are equal iff the fixed points of their defining equations are same. IR4 is a simple induction principle on the 'Size' of the argument. The induction hypothesis is not proved but assumed and from this the truth of conclusion for size = n derived.

We discuss below how the above system is used in practice. An FP program consists of a series of definitions:

Def $\varphi_i = E_i(\varphi_1, \dots, \varphi_n) \quad i = 1, \dots, n$.

where $\varphi_1, \dots, \varphi_n$ are function names and E_1, \dots, E_n are FP expressions with $\varphi_1, \dots, \varphi_n$ as variables. All the φ 's will be distinct. Let R be a set of equation

$$\varphi_i =_R E_i(\varphi_1, \dots, \varphi_n), \quad i = 1, \dots, n.$$

We distinguish these equations as they are the defining equations. Consider $P = R + S$. All the derivations will take place in the system P .

Note that in $IR3$ (Def $f_1 = E_1(f_1)$) means that $f_1 =_R E_1(f_1)$ etc. This is essential since it is possible to derive $g = E_1(g)$ where g is not the least fixed point of E_1 and hence not equal to f_1 . In proofs of equivalences presented in the next chapter we do not use the symbol $=_R$ as the context will make it apparent.

The system presented here is simple and easy to understand and use. It does not require of the user any mathematical sophistication in predicate calculus or recursion theory. Once a rule has been stated precisely it can be used just as easily as a high-school student uses arithmetic laws. Since all deductions are strictly in accordance with a set of equations, this system can be practically implemented. It can atleast be used to check the proofs mechanically. It is in this context of practical implementation that we feel Term Rewrite Systems have many advantages. They are closely related to equational theories and provide systematic ways of mechanising equational theories. See [KB 67, Hu 80, Go 80] for more details.

The next section presents some of the problems which are **yet** to be resolved for our system.

5. Problems :

Consider Def $f = * \cdot [2, f]$. This program represents a non-terminating computation. It's least fixed point is \perp , totally undefined function. But can we deduce the equation $f = \perp$ from our system of equations? There does not seem to be any way to do it.

The above remarks raise two questions. Firstly should we have to accept least fixed point theory only? Least fixed points have a well known method of implementation and also have certain theoretically interesting properties. There are a number of recursive definitions in which the function that is intuitively obvious is not the least fixed point [Ma 78]. In the above case itself f can be the function 'everywhere zero'. Thus it is not clear what should be the model of computation so that our set of equations is compatible with it.

The second question is more fundamental. It raises the question whether our system is complete. Of course, no axiomatic system similar to ours can be complete in the sense that all possible equivalences can be deduced. The existence of such a system would make halting problem of Turing machines decidable which is known to be undecidable. Cook takes a different approach to prove that Hoares axiom system is complete [Co 75], if the underlying deductive system is assumed to be non-effective. But we have not been able to proceed in that direction due to two reasons. Firstly it is not clear in our case, how to separate the deductive system. Secondly an interpretive model has to be built to show the completeness. Since we

assumed all functions to be strict construction of such a model will be complicated and completeness also will be difficult establish.

Another problem is the exact equivalence induced. Can we show that it coincides with any other known notion of equivalence? Raoult and Vuillemin [Ra 80] prove that two notions of equivalence, one introduced by a term rewrite systems and the other through recursion theory are equivalent under certain conditions. To apply this result we have to embed our equational system in a term rewrite system and then check for these conditions. Though there are some algorithms to check for these conditions, the presence of commutative axioms complicate the matter by requiring term rewrite system to operate on equivalence classes of terms rather than terms themselves. Also it is impossible to do any hand computation.

Thus the exact power of our system needs to be characterised in a rigorous way. But in the next chapter we present some proofs of equivalences which should convince that the system does have considerable power.

CHAPTER 4

EXAMPLES

In this chapter we present some of the program equivalence proofs worked out. As our first example we show the equivalence two recursively defined programs for finding the maximum element of a sequence. This example is typical use of the IR3 relating to recursively defined functions. We also present an adhoc proof of the same. Our second example is proof of McCarthy's 91 function. This example illustrates induction principle (IR4) in a general way. The third example is an exercise in program validation. A simple sorting program is defined and its salient properties are proved. These properties are fundamental to sorting and hence can be said to 'prove' that the sort program proposed is correct. The fourth example is a general theorem adopted from [WI 81]. A large number of such theorems will prove useful in carrying out proofs of equivalence.

Our presentation, in general, is quite formal. We name some of the identities derived so that they can be quoted. To shorten the proofs, we have to omit certain obvious steps. In general every equation is followed by a bracketed note giving the axioms used or other results used. Sometime we quote a result not deduced earlier, in which case it is assumed that the result is quite simple to prove.

Each proof consists of a series of results proved. Each result is in general a conditional equation and in the body of the proof of the result we do not repeat these conditions but assume that they hold. Also in most

cases we do not explicitly state the conditions relating to input data type. This is to make the proofs more readable. Also these conditions are very apparent from the context.

Lastly we point out that we use 1,2 etc., as selector functions. These can be easily defined as Hd, Hd·Tl etc. and some of their trivial properties can be established formally. The numerical constants one, two etc., are denoted as $\overline{1}, \overline{2}$ etc. There should be no confusion in this notation.

1. Example 1:

We define the following functions:

```

Def Max  = Null →  $\overline{1}$ ; Eq· [Length,  $\overline{1}$ ] → 1; Max2· [1,Max·Tl]

Def Bmax = Null →  $\overline{1}$ ; Eq· [Length,  $\overline{1}$ ] → 1; Bmax·Reduce

Def Reduce= Null →  $\overline{0}$ ; Eq· [Length,  $\overline{1}$ ] → id; Appendl·
                    [Max2· [1,2], Reduce· Tl· Tl]

Def Max2 = Ge· [1,2] → 1;2

```

Max finds the maximum by a sequential search and Bmax finds maximum by comparing neighbouring elements. Max2 finds maximum of two numbers. Our goal is to prove Bmax = Max. Since both of the functions are recursively defined we would use IR3. The proof is given as a sequence of results. For many results we would not explicitly state the input type restriction of it being a sequence of integers.

R1. (Isnumber·x) and (Isnumber·y) → Max·[x,y]=Max2·[x,y]

R2. (Isnumber·x) and (Isnumber·y) → Bmax·[x,y]=Max2·[x,y]

(R1 and R2 trivially follow from Def Max, Def Bmax and Def Max2).

R3. $\text{Not} \cdot \text{null} \cdot y \rightarrow \text{Max} \cdot \text{Append} \cdot [x, y] = \text{Max2} \cdot [x, \text{Max} \cdot y]$

Proof: $\text{Max} \cdot \text{Appendl} \cdot [x, y]$
 $= \text{Max2} \cdot [1, \text{Max} \cdot \text{Tl}] \cdot \text{Appendl} \cdot [x, y]$

R4. $B_{max} = B_{max} \cdot \text{Reduce}$

Proof: The proof uses IR1 with $p = Ge \cdot [\text{Length}, 2]$

$p \rightarrow B_{max} = B_{max} \cdot \text{Reduce}(\text{Def } B_{max})$

not·p - → Bmax·Reduce

= Null \Rightarrow $\overline{1}$; Eq. [Length, $\overline{1}$] \rightarrow 1

= B_{max} (same as above)

R5. $Ge \cdot [Length, \bar{2}] - \rightarrow Bmax \cdot Tl \cdot Tl = Bmax \cdot Tl \cdot Reduce$

Proof: $Bmax \cdot Tl \cdot Reduce$

$$\begin{aligned}
 &= Bmax \cdot Tl \cdot Appendl \cdot [Max2 \cdot [1, 2], Reduce \cdot Tl \cdot Tl] \\
 &\quad (\text{Def Reduce}; Ge \cdot [Length, \bar{2}]; A18) \\
 &= Bmax \cdot Reduce \cdot Tl \cdot Tl \quad (A3) \\
 &= Bmax \cdot Tl \cdot Tl \quad (R4)
 \end{aligned}$$

R6. $Ge \cdot [Length, \bar{2}] - \rightarrow Max2 \cdot [1, Max \cdot Tl] = Max$

Proof: follows directly from Def Max.

R7. $Max \cdot Reduce = Max$

Proof: We first use IR4 (induction) with $size=Length$ and $\bar{p} = \bar{2}$.

$$\begin{aligned}
 (\text{Base}): Ge \cdot [Length, \bar{2}] - &\rightarrow Max \cdot Reduce \\
 &= Max \cdot Appendl \cdot [Max2 \cdot [1, 2], Reduce \cdot Tl \cdot Tl] \\
 &\quad (\text{Def Reduce}; Ge \cdot [Length, \bar{2}]; A18) \\
 &= Max \cdot Appendl \cdot [Max2 \cdot [1, 2], \emptyset] \\
 &\quad (Ge \cdot [Length, \bar{2}] - \rightarrow Tl \cdot Tl = \emptyset; Reduce \cdot \emptyset = \emptyset) \\
 &= Max \cdot [Max2 \cdot [1, 2]] \quad (A5) \\
 &= Max2 \cdot [1, 2] \\
 &\quad (Length \cdot [Max2 \cdot [1, 2]] = \bar{1}; \text{Def Max}) \\
 &= Max \cdot (R1)
 \end{aligned}$$

(Induction hypothesis): $\text{Lt} \cdot [\text{Length}, \overline{n}] \rightarrow \text{Max} \cdot \text{Reduce} = \text{Max}$

(Induction Step): $(\text{Eq} \cdot [\text{Length}, \overline{n}]) \text{ and } (\text{Gt} \cdot [\overline{n}, \overline{2}]) - \rightarrow$

Max · Reduce

$$= \text{Max} \cdot \text{AppendL} \cdot [\text{Max2} \cdot [1, 2], \text{Reduce} \cdot \text{Tl} \cdot \text{Tl}]$$

(Def Reduce; Gt · [n, 2]; A18)

$$= \text{Max2} \cdot [\text{Max2} \cdot [1, 2], \text{Max} \cdot \text{Reduce} \cdot \text{Tl} \cdot \text{Tl}]$$

(Gt · [n, 2]) - \rightarrow not · null · Reduce · Tl · Tl = \overline{T} ; R3

$$= \text{Max2} \cdot [\text{Max2} \cdot [1, 2], \text{Max} \cdot \text{Tl} \cdot \text{Tl}]$$

(Lt · [Length · Tl · Tl, \overline{n}]; induction hypothesis)

$$= \text{Max2} \cdot [1, \text{Max2} \cdot [1, \text{Max} \cdot \text{Tl}] \cdot \text{Tl}]$$

(associativity of Max2; $1 \cdot \text{Tl} = 2$; A16)

$$= \text{Max} \quad (\text{R6}, \text{R6})$$

Let $p = \text{Ge} \cdot [\text{Length}, \overline{2}]$

$p - \rightarrow \text{Max} \cdot \text{Reduced} = \text{Max}(\text{Base}, \text{induction step}, \text{IR4})$

$\text{Not} \cdot p - \rightarrow \text{Max} \cdot \text{Reduce} = \text{Max}$

(DefMax; not · p - \rightarrow or · [null, Eq · [Length, $\overline{1}$]] = \overline{T} ;)

$\text{Max} \cdot \text{Reduce} = \text{Max}^{\prime}(\text{IR1})$

$$R8. \text{ Ge} \cdot [\text{Length}, \bar{2}] - \rightarrow \text{Max2} \cdot [1, \text{Bmax} \cdot T1] = \text{Bmax}$$

Proof: We use IR4 with $\bar{p} = \bar{2}$ and $\text{Size} = \text{Length}$.

$$(\text{Base}): \text{Eq} \cdot [\text{Length}, \bar{2}] - \rightarrow \text{Max2} \cdot [1, \text{Bmax} \cdot T1]$$

$$= \text{Max2} \cdot [1, 1 \cdot T1]$$

$$(\text{Def Bmax}; \text{Eq} \cdot [\text{Length} \cdot T1, \bar{1}] = \bar{T})$$

$$= \text{Bmax} \quad (1 \cdot T1 = 2; R2)$$

$$(\text{Induction hypothesis}): \text{Lt} \cdot [\text{Length}, \bar{n}] - \rightarrow$$

$$\text{Max2} \cdot [1, \text{Bmax} \cdot T1] = \text{Bmax}$$

$$(\text{Induction step}): (\text{Eq} \cdot [\text{Length}, \bar{n}]) \text{ and } (\text{Gt} \cdot [\bar{n}, 2]) - \rightarrow$$

$$\text{Max2} \cdot [1, \text{Bmax} \cdot T1]$$

$$= \text{Max2} \cdot [1, \text{Max2} \cdot [1, \text{Bmax} \cdot T1] \cdot T1]$$

$$(\text{Lt} \cdot [\text{Length}, \bar{n}] \cdot T1 = \bar{T}; \text{ induction hypothesis})$$

$$= \text{Max2} \cdot [\text{Max2} \cdot [1, 2], \text{Bmax} \cdot T1 \cdot T1]$$

$$(\text{associativity of Max2}; 1 \cdot T1 = 2)$$

$$= \text{Max2} \cdot [1, \text{Bmax} \cdot T1] \cdot \text{Reduce}$$

$$(R5; \text{Gt} \cdot [\bar{n}, \bar{2}] - \rightarrow 1 \cdot \text{Reduce} = \text{Max2} \cdot [1, 2]),$$

$$= \text{Bmax}$$

$$(\text{Lt} \cdot [\text{Length}, \bar{n}] \cdot \text{Reduce} = \bar{T}, \text{ induction hypothesis}; R4)$$

$$\text{Ge} \cdot [\text{Length}, \bar{2}] - \rightarrow \text{Max2} \cdot [1, \text{Bmax} \cdot T1] = \text{Bmax} \quad (\text{Base}; \text{induction step}; \text{IR4})$$

R9. $\text{Max-Null} \rightarrow \perp; \text{Eq} \cdot [\text{Length}, \overline{1}] \rightarrow \text{Max-Reduce}$

Proof: Follows from R7.

R10. $\text{Bmax-Null} \rightarrow \perp; \text{Eq} \cdot [\text{Length}, \overline{1}] \rightarrow \text{Max2} \cdot [\perp, \text{Bmax-} \text{T1}]$

Proof: Follows from R8.

R11. Bmax-Max

Proof: By Def Bmax, DefMax, R9, R10 and IR3.

Remark: Above we gave a straightforward application of IR3. In this case a more direct proof can be given for Bmax=Max by using IR4 as follows:

(Base): $\text{Eq} \cdot [\text{Length}, \overline{1}] \rightarrow \text{Bmax-Max}$ from Definitions.

(Induction hypothesis): $\text{Lt} \cdot [\text{Length}, \overline{n}] \rightarrow \text{Bmax-Max}$

(Induction Step): $(\text{Eq} \cdot [\text{Length}, \overline{n}]) \text{ and } \text{Gt} \cdot [\overline{n}, \overline{1}] \rightarrow$

Bmax

$= \text{Bmax-Reduce}$ (R4)

$= \text{Max-Reduce}$

$(\text{Lt} \cdot [\text{Length}, \overline{n}] \cdot \text{Reduce} = \overline{T}; \text{induction hyp.})$

$= \text{Max}$ (R7)

$\text{Ge} \cdot [\text{Length}, \overline{1}] \rightarrow \text{Bmax-Max}$ (Base; induction step; IR4)

$\text{Eq} \cdot [\text{Length}, \overline{0}] \rightarrow \text{Bmax-Max}$ (Def Bmax; Def Max)

Such proofs can be attempted when the functions are total over a datatype, in this case sequence of numbers.

2. Example 2

This example illustrates a case for which induction is used but the size function is not so straightforward as in the above proofs.

Def $F = Gt \cdot [id, 100] \rightarrow \dots [id, 10]; F \cdot F \cdot + [id, 11]$

Def $G = Gt \cdot [id, 100] \rightarrow \dots [Id, 10]; 91$

F is the well-known McCarthy's 91 function. Our goal is to prove $F = G$.

We will use induction principle IR4 with $\bar{p} = \bar{1}$ and

Size = least i such that $(n+11 \times i) > 100$

This function is easily definable in FP but we do not give a formal definition. Since we have not given axioms for arithmetic operators we assume the following properties of size. But once the axioms for arithmetic are specified these can be derived formally.

R1. $Size \cdot + \cdot [id, 11] \leq Size$

R2. $Eq \cdot [Size, \bar{0}] - \rightarrow Gt \cdot [id, 100] = \bar{T}$

R3. $Gt \cdot [Size, \bar{0}] - \rightarrow Le \cdot [id, 100] = \bar{T}$

R4. $Eq \cdot [Size, \bar{1}] - \rightarrow And \cdot [Le \cdot [90, id], Le \cdot [id, 100]] = \bar{T}$

R5. $F90 = F91 = \dots = F101 = 91$ (from Def F)

Proof:

(Base): $\text{Eq} \cdot [\text{Size}, \bar{1}] \rightarrow F = 91 \text{ (R4; R5)}$

$= G \text{ (Def } G; \text{ R3)}$

(Induction hypothesis): $\text{Lt} \cdot [\text{Size}, \bar{n}] \rightarrow F = G$

(Induction Step): $(\text{Eq} \cdot [\text{Size}, \bar{n}]) \text{ and } (\text{Gt} \cdot [\bar{n}, \bar{1}]) \rightarrow$

$F = F \cdot F \cdot + \cdot [\text{id}, 11] \text{ (R3)}$

$= F \cdot G \cdot + \cdot [\text{id}, 11] \text{ (R1; ind. hyp.)}$

$= G \cdot [\text{id}, 90] \rightarrow F \cdot + \cdot [\text{id}, \bar{1}]; F \cdot 91$

$\text{ (Def } G; \text{ A16, A17)}$

$\# G \cdot [\text{id}, 90] \rightarrow F \cdot + \cdot [\text{id}, \bar{1}]; 91 \text{ (R5)}$

$= 91$

$(G \cdot [\text{id}, 90] \rightarrow F \cdot + \cdot [\text{id}, \bar{1}]) = 91 \text{ by R3 and R5; } (p \rightarrow f; f) = f \text{ by A18, IR1}$

$= G \text{ (Def } G; \text{ Le} \cdot [\text{id}, 100] = \bar{1} \text{ by R3; A18)}$

$G \cdot [\text{Size}, \bar{1}] \rightarrow F = G \text{ (Base, induction step; IR4)}$

$\text{Eq} \cdot [\text{Size}, \bar{0}] \rightarrow F = G \text{ (R2; A18)}$

$G \cdot [\text{Size}, \bar{0}] \rightarrow F = G \text{ (by the two above equations)}$

$G \cdot [\text{Size}, \bar{0}] = \text{Isnumber (R2, R3)}$

Q.E.D.

3 Example 3:

In this section we take a different viewpoint. Our emphasis is on proving program correctness rather than program equivalence. But we reduce the problem to program equivalence. We say that if certain key properties of a program are identified and shown to hold then we can hope that the program is 'correct'. In effect these properties should specify the program. Of course for many problems the only conditions or properties that can be stated are the input output conditions in which case one may not benefit much from this.

We present here a sorting program, a very simple one, and show that it satisfies certain properties which characterise the problem of sorting.

Def SORT = Eq. [Length, \bar{l}] \rightarrow id; Append1. [Max, Sort. Rest.

[Max, id]]

Def Rest = Null. 2 \rightarrow \emptyset ; Eq. [l , l . 2] \rightarrow T_1 . 2;

Append1. [l . 2, Rest. [l , T_1 . 2]]

Def Perm = Eq. [Length. 1, Length. 2] \rightarrow Pairoff; \bar{F}

Def Pairoff = And. [Null. 1, Null. 2] \rightarrow \bar{T} ; Find. [l . 1, 2] \rightarrow

Pairoff. [T_1 . 1, Rest. [l . 1, 2]], \bar{F}

Def Ordered = Eq. [Length, \bar{l}] \rightarrow \bar{T} ; Eq. [Max, 1] \rightarrow ordered. T_1 ; \bar{F}

Def Find = Null. 2 \rightarrow \bar{F} ; Eq. [l , l . 2] \rightarrow \bar{T} ; Find. [l , T_1 . 2]

Def Max = Null \rightarrow \bar{l} ; Eq. [Length, \bar{l}] \rightarrow l ; Max2. [l , Max. T_1]

All functions are easily understood except perhaps PERM. PERM takes two sequences and checks whether they are permutations of each other or not. Find ${}^{\cdot}(x, y)$ tells whether x occurs in y or not. We have to prove the following properties.

$$\text{PERM} \cdot [\text{Sort}, \text{id}] = \bar{T} \quad (P)$$

$$\text{ORDERED} \cdot \text{Sort} = \bar{T} \quad (O)$$

R1. $(\text{Not} \cdot \text{Null} \cdot y) \text{ and} (\text{Find} \cdot [x, y]) - \rightarrow$

$$+\cdot [\bar{1}, \text{Length} \cdot \text{Rest} \cdot [x, y]] = \text{Length} \cdot y$$

Proof: We use IR4 with $\text{Size} = \text{Length} \cdot y$ and $\bar{p} = \bar{1}$

S1. $+\cdot [1, \text{Length} \cdot \text{Rest} \cdot [x, y]] = \text{Eq} \cdot [x, 1 \cdot y] \rightarrow \text{Length} \cdot y;$

$$+\cdot [\bar{1}, \text{Length} \cdot \text{Rest} \cdot [x, \text{Tl} \cdot y]]$$

(Def Rest; A17; A7; null $\cdot y = \bar{F}$)

(Base): $\text{Eq} \cdot [\text{Length} \cdot y, \bar{1}] - \rightarrow + \cdot [\bar{1}, \text{Length} \cdot \text{Rest} \cdot [x, y]]$

$$= \text{Length} \cdot y$$

(S1; Find $\cdot [x, y] - \rightarrow$ Or $\cdot [\text{Eq} \cdot [x, 1 \cdot y], \text{Find} \cdot [x, \text{Tl} \cdot y]] = \bar{T}$;

$$\text{Eq} \cdot [\text{Length} \cdot y, \bar{1}] - \rightarrow \text{Find} \cdot [x, \text{Tl} \cdot y] = \bar{F})$$

(Induction hypothesis): $\text{Lt} \cdot [\text{Length} \cdot y, \bar{n}] - \rightarrow$

$$+\cdot [\bar{1}, \text{Length} \cdot \text{Rest} \cdot [x, y]] = \text{Length} \cdot y$$

(Induction Step):

Now we use IR1 with $p = \text{Eq} \cdot [x, l \cdot y]$

$$p - \rightarrow + \cdot [\bar{l}, \text{Length} \cdot \text{Rest} \cdot [x, y]] = \text{Length} \cdot y (\text{S1}; \text{A18})$$

$$\text{Not} \cdot p - \rightarrow + \cdot [\bar{l}, \text{Length} \cdot \text{Rest} \cdot [x, y]]$$

$$= + \cdot [\bar{l}, + \cdot [\bar{l}, \text{Length} \cdot \text{Rest} \cdot [x, Tl \cdot y]]] (\text{S1}, \text{A18})$$

$$= + \cdot [\bar{l}, \text{Length} \cdot Tl \cdot y]$$

$$((\text{Find} \cdot [x, y]) \text{ and } (\text{not} \cdot p)) - \rightarrow \text{Find} \cdot [x, Tl \cdot y] = \bar{T}; \text{ Ind. Hyp.}$$

$$= \text{Length} \cdot y (\text{A7}; \text{A5})$$

$$(\text{Eq} \cdot [\text{Length} \cdot y, \bar{n}]) \text{ and } (\text{Gt} \cdot [\bar{n}, \bar{l}]) - \rightarrow$$

$$+ \cdot [\bar{l}, \text{Length} \cdot \text{Rest} \cdot [x, y]] = \text{Length} \cdot y (\text{IR1})$$

$$(\text{Ge} \cdot [\text{Length} \cdot y, \bar{l}]) - \rightarrow + \cdot [\bar{l}, \text{Length} \cdot \text{Rest} \cdot [x, y]] = \text{Length} \cdot y$$

(Base, Induction Step, IR4)

But $\text{Ge} \cdot [\text{Length} \cdot y, \bar{l}] = \text{not} \cdot \text{null} \cdot y$

R2. $(\text{Not} \cdot \text{null}) - \rightarrow \text{Length} \cdot \text{Sort} = \text{Length}$

First we state:

S2. $\text{Length} \cdot \text{sort}$

$$= \text{Eq} \cdot [\text{Length}, \bar{l}] \rightarrow \text{Length}; + \cdot [\bar{l}, \text{Length} \cdot \text{Sort} \cdot \text{Rest} \cdot [\text{Max}, \text{id}]]$$

(Def Sort; A17; A7; A1)

S3. $\text{Find} \cdot [\text{Max}, \text{id}] = \bar{T}$ (can be proved using Def Find,
Def Max and IR4)

S4. $+\cdot [\bar{1}, \text{Length} \cdot \text{Rest} \cdot [\text{Max}, \text{id}]] = \text{Length}$ (R1)

S5. $\text{Lt} \cdot [\text{Length} \cdot \text{Rest} \cdot [\text{Max}, \text{id}], \text{Length}] = \bar{T}$

(S4 and axioms of arithmetic)

Proof: We use IR4, with size=Length and $\bar{p} = \bar{1}$

(Base): $\text{Eq} \cdot [\text{Length}, \bar{1}] - \rightarrow \text{Length} \cdot \text{Sort} = \text{Length}$ (S2; A18)

(Induction hypothesis): $\text{Lt} \cdot [\text{Length}, \bar{n}] - \rightarrow \text{Length} \cdot \text{Sort} = \text{Length}$

(Induction step): $(\text{Eq} \cdot [\text{Length}, \bar{n}]) \text{ and } (\text{Gt} \cdot [\bar{n}, \bar{1}]) - \rightarrow$

$\text{Length} \cdot \text{Sort}$

$= + \cdot [\bar{1}, \text{Length} \cdot \text{Sort} \cdot \text{Rest} \cdot [\text{Max}, \text{id}]]$ (S1; A18)

$= + \cdot [\bar{1}, \text{Length} \cdot \text{Rest} \cdot [\text{Max}, \text{id}]]$

(S5; Induction Hypothesis)

$= \text{Length} \cdot \text{id}$ (S3; R1)

$= \text{Length}$ (A1)

$\text{Ge} \cdot [\text{Length}, \bar{1}] - \rightarrow \text{Length} \cdot \text{Sort} = \text{Length}$ (Base; Ind. Step; IR4)

$\text{Not} \cdot \text{Null} - \rightarrow \text{Length} \cdot \text{Sort} = \text{Length}$ ($\text{Ge} \cdot [\text{Length}, \bar{1}] = \text{Not} \cdot \text{Null}$)

R3. $\text{PERM} \cdot [\text{Sort}, \text{id}] = \text{Pairoff} \cdot [\text{Sort}, \text{id}]$ (Def PERM; R2; A18)

R4. $\text{Not} \cdot \text{Null} \rightarrow \text{Pairoff} \cdot [\text{Sort}, \text{id}] = \bar{T}$

Proof: We use IR4 with $\text{Size} = \text{Length}$ and $\bar{p} = \bar{l}$

(Base): $\text{Eq} \cdot [\text{Length}, \bar{l}] \rightarrow \text{Pairoff} \cdot [\text{Sort}, \text{id}]$

$= \text{Find} \cdot [\bar{l}, \text{id}] \rightarrow \text{Pairoff} \cdot [\bar{T} \cdot \text{Sort}, \text{Rest} \cdot [\bar{l}, \text{id}]] ; \bar{F}$
 (Def Pairoff; Def Sort; Eq $\cdot [\text{Length}, \bar{l}]$)

$= \text{Pairoff} \cdot [\bar{T} \cdot \text{Sort}, \bar{T}]$

($\text{Find} \cdot [\bar{l}, \text{id}] = \bar{T}$; $\text{Rest} \cdot [\bar{l}, \text{id}] = \bar{T}$)

$= \bar{T}$

($\text{Eq} \cdot [\text{Length}, \bar{l}] \rightarrow \text{Null} \cdot \bar{T} = \text{Null} \cdot \bar{T} \cdot \text{Sort} = \bar{T}$)

(Ind. Hyp.): $\text{Lt} \cdot [\text{Length}, \bar{n}] \rightarrow \text{Pairoff} \cdot [\text{Sort}, \text{id}] = \bar{T}$

(Ind. Step): ($\text{Eq} \cdot [\text{Length}, \bar{n}]$) and ($\text{Gt} \cdot [\bar{n}, \bar{l}]$) \rightarrow

$\text{Pairoff} \cdot [\text{Sort}, \text{id}]$

$= \text{Find} \cdot [\text{Max}, \text{id}] \rightarrow \text{Pairoff} \cdot [\text{Sort}, \text{id}] \cdot \text{Rest} \cdot [\text{Max}, \text{id}] ; \bar{F}$

(Def Pairoff; $\text{Gt} \cdot [\bar{n}, \bar{l}] \rightarrow \bar{l} \cdot \text{Sort} = \text{Max}$; $\text{Et} \cdot [\bar{n}, \bar{l}] \rightarrow$

$\bar{T} \cdot \text{Sort} = \text{Sort} \cdot \text{Rest} \cdot [\text{Max}, \text{id}]$, A16)

$= \text{Pairoff} \cdot [\text{Sort}, \text{id}] \cdot \text{Rest} \cdot [\text{Max}, \text{id}]$

($\text{Find} \cdot [\text{Max}, \text{id}] = \bar{T}$; A18)

$= \bar{T}$

(S5 of R2; Induction Hypothesis)

Not·Null- \rightarrow Pairoff·[Sort, id] = \bar{T} (Base; Ind. Step; IR4)

R5. PERM·[Sort, id] = \bar{T} (By R3 and R4)

R6. Max·Sort = l·Sort

Proof: We do not present the proof as it is again a simple application of IR4.

R7. Not·Null- \rightarrow Ordered·Sort = \bar{T}

Proof: We use IR4 with Size·Length and $\bar{p} = \bar{l}$

(Base): Eq·[Length, \bar{l}] - \rightarrow Ordered·Sort = Eq·[Length·Sort, \bar{l}] \rightarrow \bar{T} ; ...
 $= \bar{T}$ (Length·Sort = Length, A18)

(Ind.Hyp.): Lt·[Length, \bar{n}] - \rightarrow Ordered·Sort = \bar{T}

(Ind. Step): (Eq·[Length, \bar{n}]) and (Gt·[\bar{n}, \bar{l}]) — \rightarrow

Ordered·Sort

= Eq·[Max·Sort, l·Sort] \rightarrow Ordered·Tl·Sort; \bar{F}

(Def Ordered; Gt·[\bar{n}, \bar{l}]; A18, A16)

= Ordered·Tl·Sort (R6, A10, A18)

= Ordered·Sort·Rest·[Max, id] (Def Sort; A3)

= \bar{T} (S5 of R2; Induction Hypothesis)

Not·Null- \rightarrow Ordered·Sort = \bar{T} (Base; Induction Step; IR4)

4 Example 4:

We want to prove the following identity:

for $n \geq 1$ and all programs f, a, b, c :

$$[a \cdot 1, f] \cdot [b, c] = [a^n \cdot b, /f \cdot [a^{n-1} \cdot b, \dots, b, c]]$$

where $g^{n+1} = g \cdot g^n$ and $g^0 = \text{id}$

Proof: We prove by induction on n .

Basis: $n = 1$

$$[a \cdot 1, f] \cdot [b, c] = [a \cdot b, f \cdot [b, c]] \quad (\text{A14, } 1 \cdot [b, c] = b)$$

$$= [a \cdot b, /f \cdot [b, c]] \quad (\text{by A20})$$

(Ind.Hyp.): Theorem . true for $n=N$

Case $n=N+1$; Let $E = [a^N \cdot b, /f \cdot [a^{N-1} \cdot b, \dots, b, c]]$

$$[a \cdot 1, f]^{N+1} \cdot [b, c] = [a \cdot 1, f] \cdot [a \cdot 1, f]^N \cdot [b, c] \quad (\text{by Defn. of } g^n)$$

$$= [a \cdot 1, f] \cdot E \quad (\text{by induction hypothesis})$$

$$= [a \cdot 1 \cdot E, f \cdot E] \quad (\text{A14})$$

$$= [a \cdot a^N \cdot b, f \cdot [a^N \cdot b, /f \cdot [a^{N-1} \cdot b, \dots, b, c]]] \\ \quad (1 \cdot E = a^N \cdot b)$$

$$= [a^{N+1} \cdot b, /f \cdot [a^N \cdot b, \dots, b, c]] \quad (\text{by A20 and Defn. } g^n)$$

Q.E.D.

Remark: Some remarks are in order. Firstly the statement of the theorem is not an identity of FP in the true sense. This identity is in effect a schematic representation of an

infinite set of FP identities, one for each value of n , $n \geq 1$. It can be easily seen that for any particular n , the corresponding FP identity can be proved using our equations. Secondly the induction principle used is not the inference rule IR4 of FP. n is not FP variable but is part of meta-language. The induction principle belongs to the metalanguage of natural numbers.

CENTRAL LIBRARY
IIT, Kanpur
Acc. No. **82661**

CHAPTER 5

CONCLUSIONS AND SUGGESTIONS FOR FURTHER WORK

We attempted in this thesis to use the algebraic equations that relate various combining forms of FP in specifying the semantics of FP. Our approach was motivated by the approach taken in algebraic specification of data types. We have been able to provide a set of equations which is practically interesting as many program equivalence proofs can be carried out in this system of equations. We have given a fair sample of such proofs. A straight forward induction principle appears in our system as a rule of inference. The reason for not including some of the other induction principle (e.g. structural, computational etc.) is that they would increase the complexity of the system and it is not clear to us whether there will be any fundamental improvement in the power of this system. We could not give a characterisation of the power of the system of equations that is given here.

Future research can be along the following lines.

- a) Examining how completeness in Cook's sense [Co 75] can be extended to systems similar to ours. This will also involve developing an interpretive model for FP.
- b) Examining whether notions similar to sufficient completeness etc. of ADTs [Gu 78] can be defined and studied for our system.

- c) To check whether the equivalence generated by equational systems coincide with other notions of equivalence developed elsewhere. See [Rao 80].
- d) It would be interesting and useful to construct and examine an FP system in which strictness condition is removed. Perhaps one can study in which all functions are strict except the constant function. This may involve putting some restrictions on the order of substitutions.
- e) As mentioned in chapter 3, lack of general type specification facilities creates the problem of having to keep track of input types permitted. A systematic type specification mechanism would enable one to automate this. But this mechanism should fit in nicely with the rest of FP system.
- f) Practical implementation of an equational system for program transformations and deduction of program equivalence based on our experience can be attempted. In this connection term rewrite system (TRS) should prove useful as they are ideally suited to model formula manipulating systems. In this connection we refer to [Hu 80, Go 80] and the references there of.

REFERENCES:

[Ba 78] Backus, J.W.: Can programming be liberated from the Vonnumann style? A functional style and its algebra of programs.
CACM, Aug, 1978.
also as IBM research report RJ 2334.

[Co 78] Cook, S.A.: Axiomatic and interpretive semantics for an Algol fragment.
T.R. No. 79, Dept. of Computer Sciences, University of Toronto, Canada.

[Go 78] Goguen, J.A. et al: An initial algebra approach to the specification, correctness and implementation of abstract data types.
In R.Yeh, editor, Current trends in programming methodology, 1978.
Also as IBM research report RC 1876.

[Go 80] Goguen, J.A.: How to prove algebraic inductive hypothesis without induction.
LNCS 87, .

[Gu 78] Guttag, and Horning, J.J: The algebraic specification of abstract data types
in, David Gries, Ed, Programming Methodology, also in Acta informatica 1978.

[Hu 80] Huet, G.: Confluent Reductions: abstract properties and application to Term Rewriting Systems JACM, 27, 4, 1980.

[KB 67] Knuth, D and Bendix, P. : Simple word problems in universal algebras.
In J. Leech, Ed., Computational problems in Abstract algebras.

[Ra 80] Raoult, J.C., Operational and Semantic Equivalence Vuillermot, J.: between Programs
JACM 27, 4, 80.

[Wi 81] Williams, J.H.: Notes on the FP Style of programming
IBM Research Laboratory,
San Jose, California.

[Ma 78] Manna, Z. and Shamir, A.: A new approach to Recursive programs.
in A.K. James, Ed., Perspectives on Computer Science.

CENTRAL LIBRARY

82661

CSP-1982-M-KUM-6N